The Parallel Jaw Gripper: A Robotic End-Effector System

William L. Bynum
College of William and Mary

Marion A. Wise
Wise Technical Services

Nancy E. Sliwa
NASA Langley Research Center

Frank H. Willard
College of William and Mary

ABSTRACT

A robotic end-effector/sensor system has been developed and tested in the Automation and Technology Branch to enable researchers in the Intelligent Systems Research Laboratory to perform experiments in teleoperator and automated remote space operations. This end-effector, a parallel jaw gripper, has proximity and crossfire sensors for the detection of workpieces, limit and overload sensors, and manually-exchangeable fingers. The Branch has researched several variations of this end-effector, including finger-mounted force/torque sensors, automatically exchangeable ratcheting tools, and several task-specific gripper styles.

A microprocessor controller has been developed for this end-effector, with a sophisticated monitor to examine and change gains, speeds, and sensor values, and to move the grippers. This controller has been interfaced into the Teleoperator and Robotic Testbed to provide automated gripping and gripper-based sensing to the Telerobotic System Simulation currently in use in the ISRL. Four end-effector/sensor systems have been fabricated and are currently used in various configurations in the ISRL.

ACKNOWLEDGMENTS

Marion Wise had the primary responsibility for the installation and development of the end-effector and supporting hardware. In addition, Kevin Barnes, Art Hayhurst, Alan Williams, and Dave Poskevich have made substantial contributions to the development of the hardware. The initial version of the controller program for the end-effector was written by F. Wallace Harrison, with substantial subsequent contributions from Sixto Vasquez and Don Soloway. Others who have contributed to earlier versions of the end-effector software are Marion Wise, Karin Cornils, Gene Bahniuk, Frank Primiano, Taumi Daniels, and Mike Mansfield. The controller program discussed in this document was based on the earlier versions of the program. The end-effector controller program and the FORTRAN interface

between the controller program and the ISRL TRSS (TeleRobotic System Simulation) were developed by Bill Bynum and improved by Frank Willard. Bob Glover provided considerable assistance in installing the FORTRAN interface for the end-effector program into the TRSS system.

INTRODUCTION

The Automation Technology Branch of NASA Langley Research Center has been researching automation and robotic techniques for space operations since 1979. This branch has developed and currently maintains the Intelligent Systems Research Lab. The lab houses a collection of computers and robotic peripherals that have been organized into a simulation testbed for telerobotic research. This testbed is being used to investigate and develop techniques for telerobotic on-orbit operations, such as assembly of large space structures and servicing and repair of spacecraft.

The parallel jaw end-effectors being used with the robotic manipulators in the Intelligent Systems Research Lab were fabricated at Langley from designs by the University of Rhode Island [5]. This document describes the hardware of the end-effector, the associated sensors, and the control and simulation software developed for the end effector.

This end-effector, a parallel jaw gripper, has proximity and crossfire sensors for the detection of workpieces, limit and overload sensors, and manually-exchangeable fingers. The Automation Technology Branch has researched several variations of this end-effector, including finger-mounted force/torque sensors, automatically exchangeable ratcheting tools, and several task-specific gripper styles.

A microprocessor controller has been developed for this end-effector, with a sophisticated monitor to examine and change gains, speeds, and sensor values, and to move the grippers. This controller has been interfaced into the Teleoperator and Robotic Testbed to provide automated gripping and gripper-based sensing to the TeleRobotic System Simulation (TRSS) currently in use in the ISRL.

This document describes the hardware and software developed for the end effector system in the Automation Technology Branch. The hardware of the system consists of a parallel jaw end-effector,proximity and force sensors, a signal conditioner subassembly, and a dual rack mountable controller. The system software consists of the controller program, a FORTRAN interface to the ISRL TRSS system, and a program that produces a graphical simulation of the end effector.

END-EFFECTOR HARDWARE

This section describes the end-effector hardware. Figure 1 shows the end-effector mounted on a PUMA manipulator. A sketch of the end-effector is shown below in Figure 2.

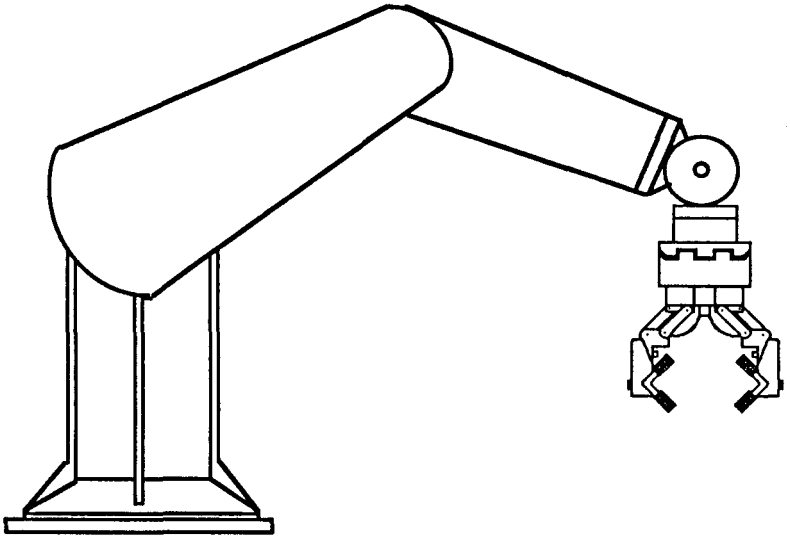

Figure 1. PUMA manipulator and end-effector

A. Mechanical Details

The body of the end-effector houses a DC torque motor. From the motor's shaft, a nylon gear drives an incremental shaft encoder and a DC tachometer, which provides position and rate feedback information respectively. A worm gear on the motor shaft drives two opposing sector gears having linkage arms. Three combinations of worm to sector gear ratios have been tested: 25:1, 50:1, and 100:1. The lower ratios lead to a low grip strength and move the jaws too rapidly. The 100:1 provides increased grip strength and slower jaw movement, but after some gear wear, the gears jam occasionally. The worm gear ratio of 100:1 is being used currently. The tachometer and shaft encoder to the motor shaft gear ratios are both 2.78:1. Each linkage arm serves as one leg of a parallelogram that moves the jaw at right angles to the motor shaft while maintaining the jaw gripping surfaces parallel to each other. The mechanical details of the end-effector are labelled in Figure 2.

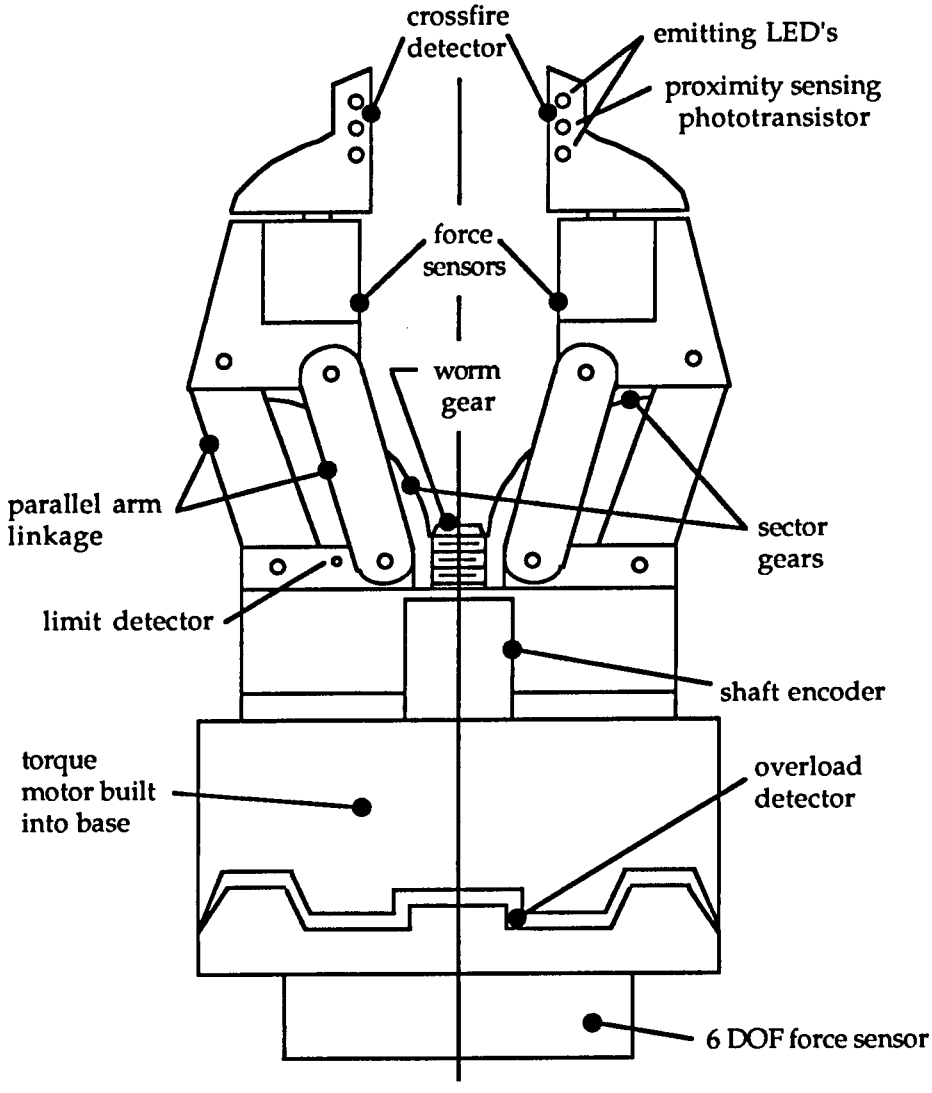

Figure 2. Components of the end-effector

Servo Components

The torque motor is an Inland Motors Model NT 2133 having a peak torque of 60 oz.-inches, and a maximum no-load speed of approximately 640 RPM.

An incremental shaft encoder, with an output rate of 250 pulses per revolution of its shaft, is geared to the motor shaft to provide position feedback. Its two TTL compatible square wave outputs are input to the microprocessor. Because it is not an absolute encoder, a position detector

located in one of the finger support blocks is used to initialize the microprocessor to zero reference position near the full open (limit) position. The limit detector consists of an infrared light emitting diode (LED) illuminating a phototransistor. The optical path between the LED and phototransistor is interrupted by a sector gear when the jaw position is within the normal operating range. An initialization subroutine in the microprocessor causes the motor to drive to the limit position which is then defined as zero position. The limit sensor is similar to the overload sensor, which is described in the "Sensors" section.

The tachometer is geared to the torque motor shaft using gears identical to those used for the encoder. Its DC output is summed with the position error signal and fed to the servo amplifier. The servo amplifier is an Inland Motors 100 watt unit with current programming resistors to limit output to a value safe for the motor. The amplifier is bolted to the frame of the controller card cage to maintain it at a safe operating temperature.

The Controller

The microprocessor on which the end-effector controller is based is the Intel 8051. In Intel terminology, the 8051 designator really stands for a family of microprocessors that includes the 8031, the 8051, and the 8751. The 8751 microprocessor has 4K of on-chip EPROM, the 8051 processor has 4K of on-chip ROM preprogrammed by Intel, and the 8031 microprocessor has no on-chip ROM or EPROM [7,8]. Although both the 8751 and the 8031 microprocessors have been used with the system, the 8031 seems to work best because of its simplicity.

The microprocessor controller uses two STD bus type printed circuit boards per end-effector. The STD bus was developed by Pro-Log and Mostek[4]. The circuit boards are located in a card cage (see the "Card Cage" section) whose front panel contains situation displays and manual controls for two independent systems. The 8031 microprocessor and its external 2716 ROM are on the CPU board, with all remaining components except the power amplifier on the second board. A 14-pin DIP header on each board allows signals to go from the CPU to the power board. This arrangement precludes the necessity for reassigning any pins on the CPU STD bus to other functions that may conflict with future use of other STD bus components. Pins for the power board have been assigned as needed and the boards keyed to prevent insertion into slots wired according to STD bus configuration.

The "A" output of the shaft encoder, and its inverse, are wired to the 8031's two interrupt pins. This ensures that both transitions (from "1" to "0" and "0" to "1") interrupt the processor, so that rotation of the encoder in ei-

ther direction can be noted. The state of the "B" encoder output at time of transition is ascertained by the interrupt routine and used to increment or decrement the position register in the 8031 as described in the software section of this report. Differences between the contents of the position register and other values are computed in accordance with input commands and the resulting error signal transmitted via the eight-bit bus to a D/A converter. The DC error signal is then summed with the rate signal from the tachometer and applied to the input terminals of the servo power amplifier. The 8031's internal circuits provide two-way serial communication with a terminal or a host computer. The serial ports are converted to RS-232 by circuits on the CPU board.

Sensors

The parallel jaw end-effector has provisions for several types of sensors in addition to the wrist mounted six-degree-of-freedom force sensor purchased for this laboratory. The eight channels of proximity sensing and a workpiece detector, or "crossfire" detector, are located in the fingertips. Each channel of the proximity sensor system consists of two infrared light emitting diodes and a phototransistor. As shown in Figure 3, the LED's illuminate an area extending a few inches from the surface of the finger. Depending upon the surface and distance, the object may be detected by the phototransistor located between the LED's, as it receives energy reflected from a nearby object. The crossfire detector uses an LED in one finger and a phototransistor in the other looking across the opening in the fingers. These are binary sensors with no provisions for obtaining range data.

All proximity and crossfire LED's are consecutively pulsed and the corresponding phototransistor sampled by circuitry in the signal conditioner unit under control of the microprocessor. Pulsing the LED's allows higher current levels by reducing average current and improves efficiency by reducing the operating temperature.

A "limit" sensor, described in the "Servo Components" section, is located in one of the finger support blocks. A similar sensor in the base, the "overload" sensor, detects relative motion between the motor housing and the base. The overload sensor output may be used with appropriate software or hardware to disengage the system when excessive forces are applied to the end-effector. A torque of zero to 30 in.lbs. will trip the overload sensor, depending on the settings of the preload adjustment springs. These two sensors operate in continuous mode using an LED-phototransistor pair looking across a short distance.

Proximity Sensor Tests

Tests of the proximity sensors were performed to determine maximum range for several reflecting surfaces. The testing was done using the orientation described in the following paragraphs.

With the tachometer in the front (facing observer), the limit sensor is in the base of the left-hand jaw (designated 1) (see Figure 2). The right-hand jaw is designated 2, and the base is designated 3. A right-hand cartesian coordinate system is then defined with the +Z-axis along the outward direction upward from the base (also known as the "approach" direction), the +Y-axis extends between the jaws toward the right jaw (called the "orientation" direction), and the +X-axis is defined by $X = Y \times Z$ (called the "normal" direction)[3]. The orientation Ya is at an angle of 60° from the Y axis of the end-effector.

The overload sensor is located in the base. The crossfire sensor's detector is located in the tip of the left jaw, and its emitter is located in the tip of the right jaw.

The proximity sensors are located on each jaw tip with the following orientation as defined by the direction of the light emitting diode:

On Jaw 1 (left jaw)	On Jaw 2 (right jaw)
+Z, +X, –X, –Ya	+Z, +X, –X, +Ya

To describe the locations of the sensors, the following convention will be used. Each sensor will be identified by a three or four alphanumeric symbol (with a minus sign) constructed as follows:

Variable sensed codes	Location of sensor codes	Direction codes
P = proximity	1 = left jaw	X
L = limit	2 = right jaw	Y
C = crossfire	3 = base	Z
O = overload		Ya = atan angle to Y-axis
F = force		+
M = moment		–
D = moment arm		

Consequently, the eleven sensors involving LED-emitters and detectors are designated as follows:

The Card Cage

The card cage front panel is hinged to provide access to the wire wrapped connectors for modifications and future expansion of the system. Presently, two separate parallel jaw end-effector systems are installed in the card cage. Each system has its own display and control on the front panel. Future possible additions would be an interface/display for the vacuum end-effector now used in the lab or an A/D converter module for the strain gauge signals.

There are three switches on the front panel of the card cage, which provide for:

- resetting the CPU,
- interrupting current to the motor, and
- manually controlling jaw position (jogging).

The motor ON/OFF switch is in series with the control winding and is after the front panel motor voltmeter. This convenience allows the operator to monitor the servo output voltage before applying power to the motor. The jog switch is a manual rate command that changes the microprocessor's commanded position. Therefore, after using the jog switch to change the jaw opening, a disagreement will exist between the actual opening and opening last received from the host computer. That error will self correct when the next position command is input.

The card cage display consists of a red LED corresponding to each of the optical sensors. The proximity sensor LED's are arranged pictorially on the front panel as they would appear to an observer looking back toward the end-effector from a workpiece. Figure 4 diagrams how the card cage appears for one end-effector.

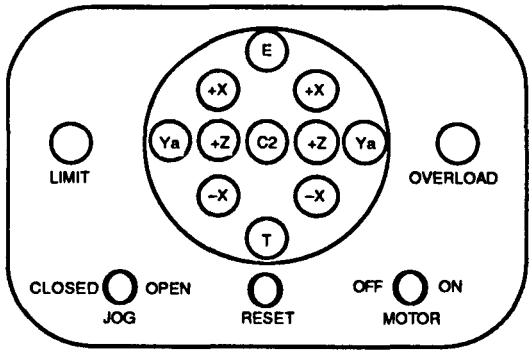

Figure 4. The front panel card cage for one end-effector

THE 8031 CONTROLLER SOFTWARE

This section discusses the structure of the 8031 program from a general, high-level point of view.

8031 program functions.

A program controlling the parallel jaw end-effector must perform several actions:

- Keep a record of the current jaw opening.
- Move the jaws, when needed.
- Record, on a regular basis, the status of the proximity, crossfire, overload, and limit sensors.
- Respond to the user's requests for jaw movement or jaw status information.

The opening of the parallel jaws is correlated to encoder counts of the shaft encoder driven by the worm screw motor through the external interrupts 0 and 1 as described in the previous section. The handlers for these two interrupts maintain a record of the current encoder counts.

Jaw movement can occur in either of two ways, a move to a commanded position or a manual "jog request" from the jog switch to open or close the jaws a small amount.

In a move to a commanded position, the controller program continually checks the actual encoder value against the target value and applies the corrective drive current to bring the actual encoder value to the target value. In a jog request, the controller program briefly applies a current to drive the jaws in the direction requested.

An extra feature available for the user is jaw self-calibration. This feature allows the user to use any set of end-effectors as the millimeter jaw gap to encoder count jaw gap ratio will be determined dynamically at system startup.

Another feature allows the user to change the speed of the jaw motor. The rate can be altered to any of 128 settings by changing the speed parameter. By changing the speed, however, the motor torque, and consequently jaw gripping force is also changed.

The status of the proximity, crossfire, and limit detectors is continually monitored by the controller program and stored in on-chip RAM. This is the information displayed on the front panel LED's.

Interaction with the user occurs through the serial port. The lower level of this interaction was described in the part of the previous section

dealing with the serial port and the associated serial interrupts. The controller program must perform the conversion from the character-level interaction with the user down to the binary level at which the controller program must operate. The controller program must also determine from the character input supplied from the user what the user wants and respond accordingly.

From the above discussion, it appears that the controller program must do several things simultaneously, which is impossible, of course, since like most people, a single processor can only do one thing at a time. This apparent concurrency is accomplished by having the processor switch rapidly from one task to another so that all of the tasks appear to progress steadily to their completion.

The 8031 Operating System.

A few words about the 8031 operating system are necessary in order to understand the 8031 controller program. The interleaving of task execution is the responsibility of the operating system kernel. It is implemented by use of one of the clock/timers as an autoload timer to generate a periodic clock interrupt. The handler for this interrupt maintains a "ready queue", a list of subroutines waiting to execute. Each subroutine in the list has an associated "schedule variable", which holds the number of clock interrupts to occur before the subroutine executes. When the clock interrupt occurs and control passes to the interrupt handler, the interrupt handler checks through the ready queue and decrements the schedule variable of each subroutine on the queue. Any subroutine whose schedule variable is zero is removed from the ready queue and made ready to execute. On termination of the handler, the subroutine removed from the ready queue executes to completion, with occasional pauses due to interrupts.

To say that this process is repeated each time a clock interrupt occurs is a slight oversimplification. The clock interrupt handler only checks the ready queue on a fraction of the times that the clock interrupt occurs. This slows the rate of checking the ready queue to a point that fits the time dynamics of the gripper hardware. Currently, the clock interrupt handler checks the ready queue every fourth clock interrupt — three clock interrupts out of four, the handler simply returns with no action. The frequency with which the clock interrupt handler checks the ready queue can be varied interactively by the user. The value for the clock interrupt handler frequency that the program uses is stored in a location of the on-chip RAM that can be modified by the user with the interactive monitor.

The frequency with which the clock interrupt handler checks the ready queue does not affect the system. Its behavior is essentially the same whether the clock interrupt value is at the minimum value of 01 (check the ready queue at every clock interrupt) or its maximum value of FF (check the ready queue only once out of every 255 clock interrupts). The explanation for this fact is that the clock interrupt frequency determines the "absolute frequency" of events in the controller system in relation to real time and not the *order* of the events or their "relative frequency". Even at the largest value for the period of the clock interrupt timer, the system events occur rapidly enough in relation to real time not to cause a problem. It is possible that if a 16-bit value were used instead of an 8-bit value to keep track of the period of the clock interrupt handler, the frequency of system events could be slowed to the point of adversely affecting system behavior.

Process Scheduling

The "schedule" subroutine of the operating system kernel places a subroutine on the ready queue and sets the schedule variable of the subroutine to the desired initial value. The subroutines in the parallel jaw controller that must perform a task repeatedly are written as "self-scheduling" routines. One of the last statements of such a subroutine would be to schedule itself into the ready queue to re-execute at some future time. This self-scheduling causes the subroutine to be invoked periodically. The self-scheduling occurs toward the end of the subroutine to avoid having two copies of the subroutine executing at the same time. At system initialization, the main program schedules all self-scheduling subroutines to get them started.

The 8031 program uses three self-scheduling subroutines. The "sensor" subroutine continually reads the proximity, crossfire, and limit sensors and displays the information by turning on the appropriate indicators on the front panel of the controller cage. The "jaw movement" routine continually checks to see whether the jaws must be moved, and if so, moves the jaws. The subroutine first checks to see if the jog switch is active and drives the jaws accordingly. If the jog switch is not active, the subroutine drives the jaws to zero the difference between the commanded position and the current position. The "jaw watching" subroutine continually compares the present jaw position with the previous jaw position, sets a "stopped" bit flag if the two positions are the same, and then calculates the error between the target position and the present position.

The schedule frequencies of the self-scheduling subroutines can be varied interactively. The schedule values for the self-scheduling

subroutines are taken from locations in the on-chip RAM; they not hard-coded into the object file. As a result, the user can vary these values using the interactive monitor described in the "Interactive Monitor" section.

This feature gives the user the flexibility to determine the optimal sizes of the scheduling intervals. This feature was necessary because the controller program did not perform correctly with the values initially chosen. Furthermore, without the ability to vary the values interactively, it was very difficult to determine what the values should be. This feature has been invaluable in getting the controller program to work as it was designed.

Tests have shown that the value of the "jaw watching" schedule variable is critical for proper behavior of the system. This routine should execute as frequently as possible. When the "jaw watching" schedule variable is large, the "stopped" bit does not accurately describe the state of the jaws because the bit can be ON even though the jaws are actually moving, and the error value between the targeted and actual positions is not valid.

The accuracy of the "stopped" bit is most crucial during initialization of the jaws. The jaws have previously been at rest, so the "stopped" bit is ON. If the "jaw watching" schedule variable is large, then as the jaws are driven open to their widest position, the "stopped" bit remains ON even though the jaws are moving, because the "jaw watching" routine, which would change the "stopped" bit, is waiting on the ready queue for its schedule variable to reach zero. The checks built into the initialization routine sense from the "stopped" bit that the jaws are stopped, even though they are actually moving, and abort the initialization prematurely.

The scheduling frequency of the "jaw watching" subroutine is also important because this subroutine computes the error between the target and actual positions. If the value of the schedule variable of the "jaw watching" subroutine is significantly larger (80 hexadecimal) than the value of the schedule variable of the "jaw movement" subroutine (0C hexadecimal), then the value of the error between the target and actual positions used by the "jaw movement" subroutine to drive the jaws is not accurate, which leads to oscillation of the jaws around the target jaw position when a jaw movement command is given. This also dictates that the "jaw watching" subroutine should execute as frequently as possible (with a self-scheduling variable of 01 hexadecimal).

The Command Interface of the Controller Program

The controller program outputs a "CMD: " prompt, after which the user may enter any of the following commands:

C	initialize the end-effector, then determine and display the range of encoder counts from the wide open to fully closed jaw positions
E	display error between target and actual positions
I	initialize the end-effector and its data structures
M	enter the interactive monitor
P	move to a commanded position
S	display current sensor values
T	display encoder counts of target position
W	display encoder counts of current position
X	change speed of jaw movement

A more extensive description of the commands and the format of the response of the controller is given in Appendix A.

The Interactive Monitor

As part of the implementation of the 8031 chip, an interactive monitor was developed. The monitor has proven quite useful during the implementation and subsequent modifications to the 8031 program. The information provided by the interactive monitor is invaluable in determining root causes for irregular or hard-to-explain program behavior.

The monitor allows the user to toggle the clock/timer interrupt on/off. By disabling the clock/timer interrupts, the user can look at the ready queue or test the sensor LED's without effecting the ready queue. If the clock/timer interrupt was not suspended, each monitor command would effect the ready queue as the ready queue is being changed by the clock/timer interrupt handler as it is being inspected.

Testing the sensor LED's is another facility of the monitor. By using the monitor, the user can test each sensor LED individually. The command, as entered by the user, actually stores a hexadecimal value between 0 and 255 in each of the two sensor bytes. It is quite useful for testing the correctness of the sensor LED's with the memory addresses.

The monitor allows the user to display and/or change the contents of an on-chip RAM location address to the value specified by the user. This facility allows the user ease-of-debugging when installing a new version of the 8031 program. The user can actually look at memory and see where the

contents differ from what was expected. In order to display the current contents of the stack pointer, SP, a special command needed to be created. This command needed to be implemented separately since the display location command uses an assembler instruction which can only access 255 memory locations, and the stack pointer's address was not among these addresses.

Appendix B contains a detailed listing of the monitor commands.

END-EFFECTOR INTERFACE WITH RTCOMMON.

The ISRL parallel jaw end-effector must be able to store its data in a way that is communicable with the rest of TRSS (TeleRobotic System Simulation), the robotics system at ISRL. The TRSS system currently operates two Unimate PUMA arms with a planned expansion in the near future. The system uses a FORTRAN "COMMON" block data structure named RTCOMMON (RealTime COMMON block) to communicate information with the rest of the system. The general format of this COMMON block and the end effector data stored in the COMMON block are discussed in later sections.

The two main programs which use RTCOMMON are OPERATOR INTERFACE and IEEECOM . These programs are discussed in the next two sections.

RTCOMMON's relationship to the VAX 11/750

The RTCOMMON data structure resides in a special portion of memory on a VAX 11/750 called "shared common memory". The shared memory location on the VAX is a permanent portion of the memory of the VAX specifically set aside for RTCOMMON. The memory is set aside so that the multiple processes that run concurrently when the TRSS system is up have a quick means by which to communicate. This scheme provides the fastest way for multiple processes to communicate with each other, and acts as a "global" COMMON block. Any process executing on the VAX may link to the RTCOMMON memory area and modify the contents of the fields in RTCOMMON. Although this open environment can be dangerous, the researchers in the ISRL environment co-operate with each other to prevent "false modification" problems.

OPERATOR INTERFACE and RTCOMMON

The person running the system sends commands through a high level interface called OPERATOR INTERFACE. This command interface is completely documented elsewhere [1]. The commands that are relevant to end-effector control will be summarized briefly here.

One command that the operator can issue from OPERATOR INTERFACE is "GAP", which specifies the gap size in millimeters. Because of the large number of different jaw types that can be used with the end-effector, this jaw gap is measured from the "heel" of the jaw mounting fitting (see Figure 5 below).

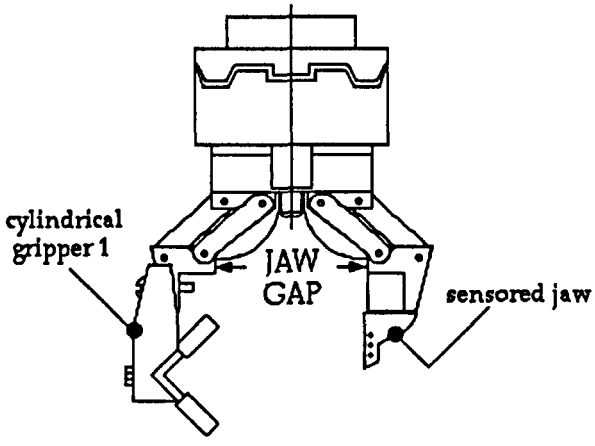

Figure 5. Measurement of jaw gap

This command allows the user to open or close the end-effector to a desired position for object manipulation. The desired position is stored in the RTCOMMON data structure, where it will be detected by IEECOM. Whenever the gap command is issued, the user must also issue a "WAIT TIME 0.5" command. This wait sequence gives the end-effector time to move to whatever the new position is.

Another operator command is the "SENSORS" command. The sensors command toggles the collection and storage of end-effector sensor information in the RTCOMMON data structure. Included among the toggled sensor information is "proximity" information which detects the presence of a nearby object, "crossfire" information which detects the presence of an object between the end-effector jaws, "base-overload" information which detects excessive forces between the motor housing and base, and "limit information", which detects the maximum opening of the end-effector.

The third operator command is "INITIALIZE". This command returns the end-effector to its state of initialization, which includes moving the end-effectors to their maximum open position. A boolean field in RTCOMMON is updated with this information and the end-effector is subsequently initialized. "INITIALIZE" can also be used to release a grasped object.

Additionally, the user may specify a file which acts as a "script" of actions. OPERATOR INTERFACE will perform the script actions sequentially.

IEECOM and RTCOMMON

IEEECOM (IEEE COMmunications) provides communication capability between the OPERATOR INTERFACE program and the various peripheral actuator devices in the TRSS. The IEEECOM process loops continuously, looking for changes in control variables in the RTCOMMON block made by one of the routines in the OPERATOR INTERFACE program requesting an action from one of the peripheral devices in the system. When such a change occurs, the proper subroutine of IEEECOM is invoked to perform the communication with the peripheral device necessary to accomplish the desired action.

The general format of RTCOMMON

In the TRSS, the user can move the end-effector by using either the "INITIALIZE" or "GAP" commands in the OPERATOR INTERFACE program. The initialize command moves the end-effector to the "open" position. The gap command specifies an absolute opening, in millimeters, for the jaw to open to. The "SENSOR" command in OPERATOR INTERFACE toggles the collection of sensor information in the RTCOMMON data structure each time through the main event loop in the IEEECOM program. This command does not present the user with sensor data directly. To obtain this information, the user must access the part of the RTCOMMON data structure relating to the end-effector in the FORTRAN source code that the user writes.

RTCOMMON contains a data structure to describe each manipulator. The data structure contains entries for the output mode, the control mode, a power-on indicator, the geometric data for the arm, and records for the control station information, the vision system, the force-torque sensor, the end-effector, the joint velocities, the resolved-rate inputs, and the homogeneous transform. The exact FORTRAN code for the RTCOMMON data structure is shown in its entirety in Appendix C. The next section describes the part of the RTCOMMON structure having to do with the end-effectors.

End-effector Data stored in RTCOMMON

The data stored in RTCOMMON for each end-effector (the "pjg_end_effector" structure) includes:

- a flag to initialize the parallel jaw data ("initialize").

- a flag to nullify the drive voltage sent to the motor ("null_motor").
- the current opening of the jaw in encoder counts ("jaw_gap").
- the error between the current jaw opening and the desired position ("error_gap").
- the desired jaw position ("target_gap")
- the current rate at which the end-effector will move ("speed").
- a sensor data record ("sensor_data").
- a ratchet data record ("ratchet_data").

The sensor data record consists of:

- a "desired" boolean which is set to true when the user wants sensor information.
- two 32-bit integer fields which store, bit-by-bit, whether the proximity, crossfire, overload, or proximity sensors are on or off ("xfire_ovl_lim" contains the first three sensors and "proximity" contains the rest). The bit assignments for these two words are as follows:

bit #	proximity word	xfire_ovl_lim word
0	Left jaw, +Ya	Crossfire
1	Left jaw, +Z	Overload
2	Left jaw, +X	Limit
3	Left jaw, -X	Unused
2	Left jaw, +X	Limit
3	Left jaw, -X	Unused
4	Right jaw, -Ya	Unused
5	Right jaw, +Z	Unused
6	Right jaw, +X	Unused
7	Right jaw, -X	Unused
8-31	unused	Unused

The ratchet record contains:

- a boolean which is set to true if the user wishes to use the ratchet ("desired").
- the gap at which the ratcheting is centered ("center_gap").
- the total number of degrees which the ratchet is to sweep ("sweep_degrees").
- the rate at which the ratcheting is to be done ("rate").

The actual FORTRAN declaration of the RTCOMMON data structure is contained in Appendix C.

CONCURRENT SIMULATION OF A PARALLEL JAW END-EFFECTOR.

This section describes a system of programs developed to provide a graphical simulation of the parallel jaw end-effector which executes concurrently with the program controlling the end-effector. The simulation system generates a graphical image of the end-effector that moves in approximately real time in response to commands issued by the controlling program. The commands that the simulation system accepts are identical to the commands the actual end-effector accepts. A sketch of the representation of the end-effector on the VS11 graphics terminal is shown below in Figure 6.

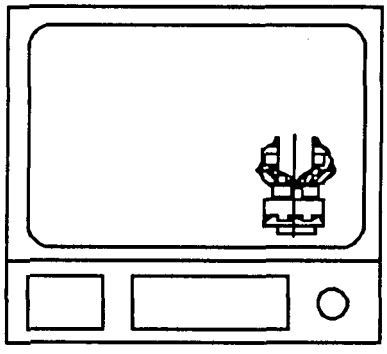

Figure 6. The graphics display of the end-effector simulator

This system provides the user with an environment that can be used to develop the command/response protocols needed for the end-effector without the necessity of scheduling usage of the end-effector and the manipulator to which it is mounted. In addition, the simulation system can be used when the actual hardware, the end-effector, the manipulator, or the OPERATOR INTERFACE of the TRSS system, are inoperative.

The simulation system has been tested with a LISP program using the command format of the DAISIE robotics control system [2], developed in the Intelligent Systems Research Laboratory.

Structure of the Simulation System

The simulation system is composed of three main parts:

- the controlling LISP program

- a collection of FORTRAN subroutines imported by the LISP program to initiate execution of, and then to communicate with, the third part of the system.
- the FORTRAN program for the overall control of the communications and graphics subroutines which produce the simulation of the end-effector.

To begin a simulation session, the user must load the file of LISP control functions into LISP and initiate the master function at the top level. This function begins an initialization sequence that imports the necessary FORTRAN subroutines into the LISP environment.

One of these imported FORTRAN subroutines is called to start the simulation. This subroutine initializes the graphics display, the interprocess mailboxes, and event flag clusters, and then creates, as a separate process, the FORTRAN program that generates the graphics and simulates the end-effector. Two mailboxes are created, one for commands sent from the controlling LISP program to the simulation program and the other for responses going the other way. Because VMS has a "wait-for offspring" rule, this initialization subroutine executes to completion, but cannot terminate until the process that it created, the end-effector simulation process, terminates. This ensures that the event flag cluster and the mailboxes that it creates continue to exist. Creating "permanent" mailboxes is another alternative, but the method chosen requires less privilege from VMS. At termination of the initialization subroutine, control returns to the calling LISP program, unlocking the keyboard and allowing additional LISP functions to be invoked.

The command and response information exchanged by the LISP controlling program and the actual end-effector has the format of a five character field accompanied by a six-vector of real numbers. Each device in the DAISIE system communicates using this same command/response format: this is one of the strengths of the design of the system. The character field gives the source and destination of the command or response, as well as the basic command or response. The six-vector of real numbers contains modifying information. For example, in the case of the move command, this information would include the type of move (ratchet, positional, etc.), and the end condition desired, along with any necessary parameters, such as rate gains, desired jaw gap or maximum jaw force at termination. In the case of a response to a status request, the six-vector contains the current and target jaw gaps, position and rate gains, and information from proximity and cross-fire detectors located on the end-effector.

As a convenience to the user, each of the commands sent to the end-effector simulator is embodied as a LISP function. This relieves the user of the burden of real-time parenthesis matching and of having to memorize the exact command formats and the encoding of their fields. Each of these LISP functions transforms its command into the DAISIE format and invokes one of the imported FORTRAN subroutines to place the command message into the command mailbox and set an associated event flag. The process simulating the end-effector is waiting on this event flag. When the flag is set, the process retrieves the command from the mailbox and acts on it.

The move and initialize commands are enqueued by the simulation program for subsequent action, whereas the quit, status request and debug commands receive an immediate reply. Commands involving movement of the end-effector are enqueued because, in general, they require a longer time to complete than a status request. With this program structure, it is possible for the LISP driver program to issue a sequence of move commands and then to monitor their progress through a series of status requests.

When a command has been completed, the simulation process sends a reply to the LISP program by placing an appropriately formatted message in the reply mailbox and setting the associated event flag. At present, the simulation program assumes that all commands complete successfully, although random failure reports could be easily introduced into the simulation program in the future.

The debug command is not really a part of the DAISIE command structure, but was added to be able to obtain debugging information during the development of the simulation system. With it, the user can toggle the display of four different types of debugging information, event flag behavior, interprocess communication, end-effector control, and graphics information.

Advantages of the simulation system

Perhaps the most important feature of the simulation system is that both the data structures and the calling structures used in the simulation are the same as the data structures used in the TRSS. This feature of the simulation facilitates the integration testing of new features to the TRSS, since no translation of data structures or subroutine calling structures is necessary. Additionally, the simulation allows accurate testing of programs without tying up the end-effector and manipulator which are scarce resources.

Details of Usage

To start the simulation, the user must load the LISP functions driving the simulation of the end-effector into the LISP interpreter. Then the user must type

```
(pjg)
```

to initialize the simulation and graphic software. The user is then presented with the prompt:

```
Enter d h(elp) i m n q s x
```

The explanation of these commands is given help screen, displayed in response to the "h" command:

```
========  PARALLEL JAW GRIPPER SIMULATOR =======
                List of Commands
      d     toggle the display of debug information
      h     display this screen
      i     reset gripper to initial condition
      m     send a move command to the gripper
      n     null command (no-op)
      q     return to LISP interpreter
      s     send a status request to the gripper
      x     exit to VMS command level
```

The most of the commands shown either correspond closely to the commands of the OPERATOR INTERFACE program described in the previous section are self-explanatory. When the user issues the "m" command, the graphical simulation of the end-effector moves in approximate real time to the commanded position.

TOOLS FOR THE PARALLEL JAW END-EFFECTOR

Several tools have been designed and fabricated for use with the parallel jaw end-effector, along with several replaceable sets of jaws. The tools have been designed to permit quick-change by programmed action of the manipulator arm and end-effector, while the jaws are typically changed manually on a task basis.

Quick-Change

The tool quick-change uses a pair of spring-loaded retaining arms to lock the tool onto the square shaft of the tool pickup. After use, the tool can be returned to the rack by rotating the holder to a particular orientation such that inclined planes on the rack open the spring-loaded arms, thus permitting the pickup shaft to be withdrawn from the tool and another tool selected. By rounding the end of the square shaft and building some

compliance into the base of the tool rack, accuracy requirements for control of the arm's position were reduced to levels that could be handled either by a human operator or by off-line programming.

Ratchets

The most useful of the tools has been the quarter-inch tool driver. One version uses motion of robot wrist to rotate the tool, in which case a ratchet is necessary because of the limited available angular rotation of the robot wrist. Sensors in the end-effector jaws or on the robot wrist have been used to control torques applied to the workpiece.

A version of the ratchet was designed to make use of the end-effector's internal motor to turn rotary tools. The ratchet was necessary in this case because the mechanism controlling end-effector jaw opening remained coupled to the motor. Thus, motor rotation over more than part of a turn would cause large changes in jaw opening beyond that required for contact with the ratchet body and proper operation of the ratchet's direction reversing lever. This version was fabricated and assembled and although it appeared to function as planned, appropriate software was not generated to evaluate the device fully. A sketch of the end-effector with the ratchet mounted is shown in Figure 7.

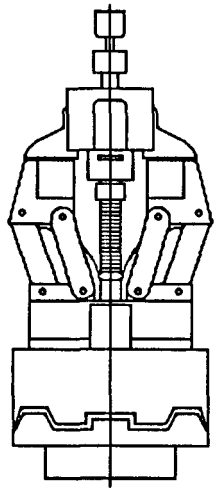

Figure 7. Ratchet tool

Alternate Fingers

The parallel jaw end-effector has worked well for ordinary operations. However, when assembling a lattice-like structure such as the proposed space station, the end-effectors had a problem with holding onto the pipe. This "slippage" problem displayed the need for another end-effector tool to be created: an end-effector with alternate fingers. In designing the alternate fingers, it was found that the rounded surface of the casing holding the rounded alternate finger structure created more surface area for the

manipulator to touch the object being gripped. This added surface area contact aided in preventing slippage.

Figure 8 shows alternate fingers developed for the end effector. The leftmost and third fingers were designed with the goal of handling cylindrical objects easily. The second finger contains the finger force and proximity sensors. The rightmost finger was designed for a small-radius handle specific to a space station application.

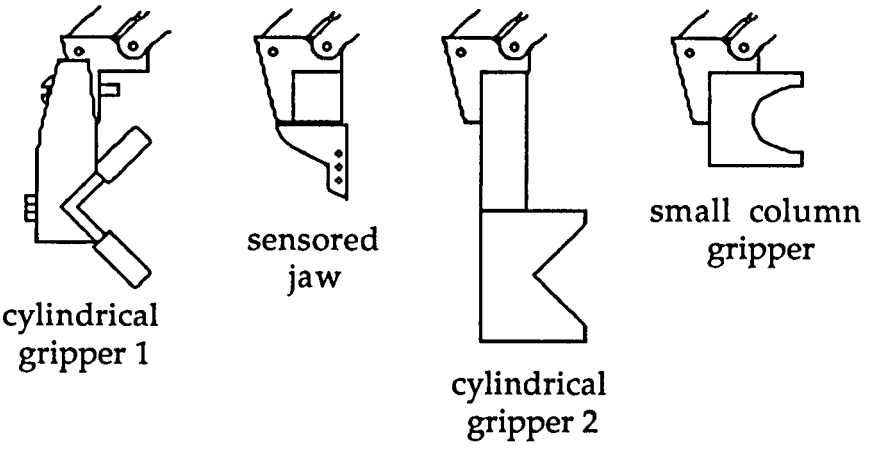

Figure 8. Alternate fingers developed for end-effector

Other Tools

Wirecutters, forceps and pliers have also been designed and fabricated for the end-effector. These tools were designed to be operated by the sensored fingers of the end-effector fitting into the openings at the base of each tool. These tools have not been fully evaluated. A sketch of the pliers is shown in Figure 9.

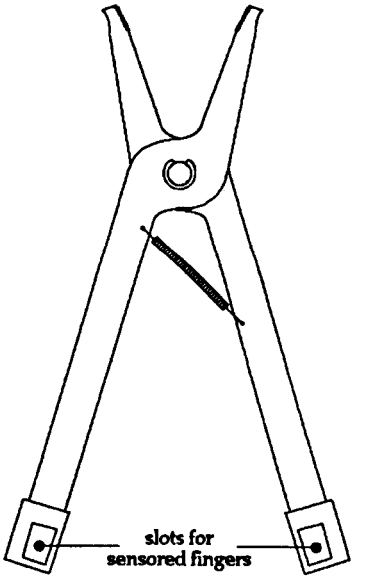

Figure 9. End-effector pliers

CONCLUSIONS

The parallel jaw end-effector system described here represents the effort of a number of people over a period of several years. The system is versatile, reliable, easily modifiable, and offers diverse functionality. In addition to the purposes for which it was designed, the software of the system provides the user with a wide variety of debugging capabilities. The end-effector system provides indispensible gripping and gripper-based sensing to the Telerobotic System Simulation research currently being conducted in the Intelligent Systems Research Laboratory by the personnel of the Automation and Technology Branch.

Appendix A
8031 Controller Commands

The values entered and returned from all controller commands are in hexadecimal, with the single exception of the current state of the sensors returned by the "S" command in binary.

Controller Commands

C Display the range of encoder counts from the wide open to the fully closed positions. This command initializes the gripper as in the "I" command below, closes the jaws until they stop, displays the corresponding encoder count value, and re-initializes the gripper. The value displayed is slightly more than the actual total range of encoder counts because of looseness in the jaw linkage and compliance of the material used in the jaw surfaces. If either initialization of the gripper fails, control passes to the interactive monitor (see the "M" command below).

E Display error between target and actual positions. The error value is kept in excess-128. Since the error value is shown in hexadecimal, a value of "0080" corresponds to zero error, a value of "008F" corresponds to an error of 15 (the actual position is 15 encoder counts more than the target value), and a value of "007A" corresponds to an error of –6. The error value is always equal to the actual position (the value displayed by the "W" command) minus the target position (the value displayed by the "T" command).

I Initialize the gripper and its data structures. This command initializes the gripper by initializing the on-chip RAM location dealing with encoder counts and then driving the jaws to the wide-open position. The wide-open position of the jaws is sensed by the limit sensor. The current value of this sensor is stored in the "limit" bit. If the initialization procedure notices that the jaws have stopped (the "stopped" bit is on) before the "limit" bit comes on, the initialization is aborted with an error message, and control is passed to the interactive monitor so that the user can determine the cause of the initialization failure (see the "M" command below).

M This command enters the interactive monitor. The command line prompt changes from "CMD: " to "* ". The following commands are accepted:

Monitor Command	Description
C	change contents of a location in on-chip RAM
D	display contents of a location in on-chip RAM

I	toggle clock/timer interrupts
L	test sensor LEDs on card cage front panel
N	display contents of next location in on-chip RAM
Q	quit the monitor and return to main command level
S	display contents of SP, the stack pointer

Any other characters are ignored. Appendix B contains more detailed description of the monitor commands.

P Move to a commanded position. This command accepts a target encoder value from the user, stores it in the on-chip RAM, and sets the "drive" bit so that the "jaw movement" routine will start driving the worm screw motor to zero the difference between the current encoder value and the target value.

Encoder counts range from "0000" when the jaws are wide open to the value of approximately "D600" when the jaws are closed. Encoder counts are always non-positive and are represented in two's complement notation. Consequently, a value of "D600" really stands for the two's complement negative of "3A00". Therefore, the commands "PE47A" and "P–1B86" have the same effect, since "E47A" is the two's complement representation of the negative of "1B86". Either form of the command will be accepted by the command interpreter.

A comment is needed about use of the "P" command. The "P" command has been improved in this version of the program. With the two's complement convention used for the number of encoder counts, any positive number of encoder counts can never be achieved. In the old program, supplying a positive target would cause the jaws to be driven closed and the position encoder counts would become increasingly negative. This resulted in an unrecoverable error situation since the encoder counts could never become positive. The new program rejects a positive encoder target as an invalid command.

S Display current sensor values. This command displays the current sensor values as a 16-bit binary string. These values are stored in locations 2E and 2F of the on-chip RAM. The leftmost eight bits of the binary string are the bits of location 2F and the rightmost eight bits are the bits of location 2E. The bits have the following meanings (bit 15 is the leftmost bit and bit 0 is the rightmost bit):

Bit Position	LED Location
15..11	unused – bit might be either 0 or 1

10	limit
9	overload
8	crossfire
7	right jaw, –X
6	right jaw, +X
5	right jaw, +Z
4	right jaw, –Ya
3	left jaw, –X
2	left jaw, +X
1	left jaw, +Z
0	left jaw, +Ya

The bit in the sensor bit string is "0" if and only if the corresponding sensor is ON. For example, if, in response to the "S" command, the user receives the bit string:

1111101101101101

then this would indicate that the following detectors are *on* (proceeding from left to right):

crossfire (leftmost 0)
right jaw, –X
left jaw, +Z (rightmost 0)

T Display target position. This command shows the last commanded position in encoder counts. When the jog switch is used to open or close the jaws, the target position is kept equal to the actual position, so that the jaws will not be moved back to the previously targeted position by the "jaw movement" subroutine.

W Display current position. This command shows the encoder counts corresponding to the current jaw opening.

X With this command, the user can change the speed at which the jaws open and close. When a combination of worm and sector gears is used in an end-effector with a significantly higher gear ratio than the ratio used in the original design, the maximum speed of the worm drive motor has to be reduced because the jaws move too rapidly. With the original worm/sector combination, a speed value of "7F" is used, but with the newer high-ratio worm/sector combination, a speed of "33" seems to move the jaws at approximately the same speed as the original gearing.

Appendix B
8031 Controller Monitor Commands

The commands accepted by the interactive monitor of the 8031 controller are listed below. All other character input is ignored.

C addr value

Change the contents of on-chip RAM location "addr" ($0 \leq addr \leq 7F$) to "value". The address and new value stored at the address are displayed after the storage takes place.

D addr

Display the contents of on-chip RAM location "addr" ($0 \leq addr \leq 7F$).

I

Toggle the clock/timer interrupt on or off. This command is needed to be able to look at the ready queue or test the sensor LEDs. Disabling the clock/timer interrupt suspends all changes to the ready queue so that the user can use the "D" monitor command to inspect the ready queue and the associated schedule variables. If the clock/timer interrupt is not disabled, then use of the "D" command to inspect the ready queue leads to confusing results, since the ready queue is being changed by the clock/timer interrupt handler as it is being inspected.

L bit-pattern

Test sensor LEDs. This command allows the user to test each sensor LED individually. The full word (two-byte) "bit-pattern" is stored at the two on-chip RAM locations used to store the sensor information. The proximity sensor bits are stored at location 2D and the crossfire, overload, and limit bits are stored at location 2E. This command stores the full word bit pattern entered by the user at locations 2D and 2E and then displays the contents of location 2D. The action of displaying the contents of the location just stored was added to make sure that the intended value for the sensor LEDs was actually being stored at locations 2D and 2E. The following example of use of this command may be helpful. In the interaction shown below, the typing done by the user is underlined.

```
*I
*L FBFE 2D FE
*N 2E FB
```

The user has stored the bit pattern "FBFE" (in hexadecimal) or "1111 1011 1111 1110" (in binary) in the sensor bytes of the on-chip RAM. The *I* command was necessary to suspend the clock/timer interrupts, so that the self-scheduling "sensor" subroutine will not replace the values entered with the current sensor values. The value "FE" is stored at location 2D and the value "FB" is stored at the location 2E (the low order byte is stored at the lower address and the high order

byte is stored at the higher address). The "FE" of the pattern turns on the crossfire LED, and the "FB" turns on the LED corresponding to the +X detector on the right jaw (see the main command level "S" command for a description of the locations of the sensor bits). The LED is ON if and only if the corresponding bit is zero. You will notice that, as a side-effect of the command, the contents of location 2D are displayed. If, as the user has done here, you want to see the contents of the next byte (the next sensor byte containing the limit, overload, and crossfire bits), you can use the monitor "N" command to display the contents of the byte at location 2E. The following bit patterns can be used to test each LED on the front panel of the controller cage:

Bit Pattern	LED Location
FF7F	–X, left jaw
FFBF	+X, left jaw
FFDF	+Z, left jaw
FFEF	–Ya, left jaw
FFF7	–X, right jaw
FFFB	+X, right jaw
FFFD	+Z, right jaw
FFFE	+Ya, right jaw
FBFF	limit
FDFF	overload
FEFF	crossfire
F800	all LEDs ON
0000	all LEDs ON
FFFF	all LEDs OFF
07FF	all LEDs OFF

N

Display the next memory address and its contents. The address displayed is the address next to the most recently referenced address by the "C" or "D" commands. When the monitor is entered, the default memory address is set to the current value of the stack pointer, SP.

Q

Quit the monitor and return to the main command loop.

S

Display the current contents of the stack pointer, SP. Even though the stack pointer is kept at location 81 in the on-chip RAM, the "D" command cannot be used to display its value. The "D" command is implemented using the MOV instruction with register-indirect addressing, and the 8031 chip restricts the range of this instruction to the

values between 00 and 7F. The stack pointer cannot be accessed by the "D" instruction since its address is outside of this range.

Appendix C
RTCOMMON Data Structure

```
c
c %%%%%       RTCOMMON.DEF -- Realtime data structures
     %%%%%
c
c
C           DATA STRUCTURE FOR INLAB DUAL PUMA SIMULATION
C
C DEFINE ALL DATA TYPES
C
     STRUCTURE     /LOCATION/
       UNION
         MAP
           REAL    X,Y,Z
         ENDMAP
         MAP
           REAL    LOCATION(3)
         ENDMAP
       ENDUNION
     ENDSTRUCTURE

     STRUCTURE     /ORIENTATION/
       UNION
         MAP
           REAL    RX,RY,RZ
         ENDMAP
         MAP
           REAL    ORIENTATION(3)
         ENDMAP
       ENDUNION
     ENDSTRUCTURE

     STRUCTURE     /EULER_VECTOR/
       UNION
         MAP
           REAL    X,Y,Z
           REAL    RX,RY,RZ
         ENDMAP
         MAP
           REAL    TRAN(3)
           REAL    ROT(3)
         ENDMAP
         MAP
           RECORD  /LOCATION/    TRANSLATION
           RECORD  /ORIENTATION/     ROTATION
         ENDMAP
         MAP
           REAL    E_VEC(6)
         ENDMAP
       ENDUNION
     ENDSTRUCTURE

     STRUCTURE     /HOMO_TRAN_MATRIX/
       UNION
```

Appendix C
RTCOMMON Data Structure

```
        MAP
          REAL    N_VECTOR(4)
          REAL    S_VECTOR(4)
          REAL    A_VECTOR(4)
          REAL    P_VECTOR(4)
        ENDMAP
        MAP
          REAL    NX,NY,NZ,NW
          REAL       SX,SY,SZ,SW
          REAL       AX,AY,AZ,AW
          REAL       PX,PY,PZ,PW
        ENDMAP
        MAP
          REAL    HTRAN(4,4)
        ENDMAP
         ENDUNION
       ENDSTRUCTURE

     STRUCTURE     /JOINT/
       UNION
         MAP
           REAL    JNT1
     2         ,JNT2
     3         ,JNT3
     4         ,JNT4
     5         ,JNT5
     6         ,JNT6
     7         ,JNT7
         ENDMAP
         MAP
           REAL    JNT(7)
         ENDMAP
       ENDUNION
     ENDSTRUCTURE

C ------------------------------------------------------------------
--
     STRUCTURE     /POSITION_VECTOR/
       RECORD /EULER_VECTOR/    POS
        ENDSTRUCTURE

     STRUCTURE     /VELOCITY_VECTOR/
       RECORD /EULER_VECTOR/    VEL
        ENDSTRUCTURE

     STRUCTURE     /ACCELERATION_VECTOR/
       RECORD /EULER_VECTOR/    ACCEL
        ENDSTRUCTURE

     STRUCTURE     /JERK_VECTOR/
       RECORD /EULER_VECTOR/    JERK
        ENDSTRUCTURE
```

Appendix C
RTCOMMON Data Structure

```
      STRUCTURE     /DATA_VECTOR/
        RECORD /EULER_VECTOR/     DATA
          ENDSTRUCTURE

      STRUCTURE     /GAIN_VECTOR/
        RECORD /EULER_VECTOR/     GAIN
          ENDSTRUCTURE

      STRUCTURE     /COMMAND_VECTOR/
        RECORD /EULER_VECTOR/     COM
          ENDSTRUCTURE

C -----------------------------------------------------------------
--

      STRUCTURE     /P_BUFFER/
        UNION
          MAP
            BYTE    BYTE_BUFFER(64)
          ENDMAP
          MAP
            INTEGER*2    I2_BUFFER(32)
          ENDMAP
          MAP
            INTEGER*4    I4_BUFFER(16)
          ENDMAP
          MAP
            REAL*4  R4_BUFFER(16)
          ENDMAP
        ENDUNION
      ENDSTRUCTURE

      STRUCTURE     /ARM_DATA/
        RECORD /JOINT/            ANGLE
        RECORD /JOINT/            VELOCITY
        RECORD /JOINT/            TORQUE
        RECORD /JOINT/            COM_ANGLE
        RECORD /JOINT/            COM_VELOCITY
        RECORD /JOINT/            COM_TORQUE
        RECORD /JOINT/            RESET_ANGLE
        RECORD /JOINT/            OFFSET
        RECORD /JOINT/            MOTOR_CURRENT
        RECORD /COMMAND_VECTOR/   POSITION
        RECORD /LOCATION/         TRANS_DUMMY
        RECORD /ORIENTATION/           EULER
      ENDSTRUCTURE
C *** NOTE NEED TO CHANGE EULER TO EULER_VECTOR, NOT CHANGE NOW, SO
NOT
C     TO DESTORY
C     THE STRUCTURE OF RTCOMMON. ***

      STRUCTURE     /CONTROL_STATION/
        RECORD /JOINT/       JOINT_BY_JOINT
```

Appendix C
RTCOMMON Data Structure

```
  REAL        ATTITUDE(3)
1         ,TRANSLATION(3)
3         ,SIX_DOF_CTRLR(6)
4         ,GAIN
ENDSTRUCTURE


STRUCTURE     /VISION_DATA/
  RECORD /DATA_VECTOR/     RAW
  RECORD /DATA_VECTOR/     PROCESSED
  LOGICAL     ACQUISITION
1         ,VISION_CONTROL
2         ,SWITCH_CONTROL
ENDSTRUCTURE


STRUCTURE     /HOMO_TRANS/
  RECORD /HOMO_TRAN_MATRIX/     CRF_TO_WORLD
  RECORD /HOMO_TRAN_MATRIX/     WORLD_TO_CRF
  RECORD /HOMO_TRAN_MATRIX/     MRF_TO_WORLD
  RECORD /HOMO_TRAN_MATRIX/     WORLD_TO_MRF
  RECORD /HOMO_TRAN_MATRIX/     BASE_TO_WORLD
  RECORD /HOMO_TRAN_MATRIX/     WORLD_TO_BASE
  RECORD /HOMO_TRAN_MATRIX/     MRF_TO_CRF
  RECORD /HOMO_TRAN_MATRIX/     CRF_TO_MRF
  RECORD /HOMO_TRAN_MATRIX/     WRIST_TO_BASE
  RECORD /HOMO_TRAN_MATRIX/     BASE_TO_WRIST
  RECORD /HOMO_TRAN_MATRIX/     MRF_TO_WRIST
  RECORD /HOMO_TRAN_MATRIX/     WRIST_TO_MRF
  RECORD /HOMO_TRAN_MATRIX/     TARGET_TO_LENS
  RECORD /HOMO_TRAN_MATRIX/     LENS_TO_TARGET
  RECORD /HOMO_TRAN_MATRIX/     LENS_TO_WRIST
  RECORD /HOMO_TRAN_MATRIX/     WRIST_TO_LENS
  RECORD /HOMO_TRAN_MATRIX/     DEFAULT
  RECORD /HOMO_TRAN_MATRIX/     WRIST_TO_ANGLE
  RECORD /HOMO_TRAN_MATRIX/     ANGLE_TO_WRIST
  RECORD /HOMO_TRAN_MATRIX/     DEFAULT_EXT
  RECORD /HOMO_TRAN_MATRIX/     WRIST_TO_FS
  RECORD /HOMO_TRAN_MATRIX/     FS_TO_WRIST
ENDSTRUCTURE


STRUCTURE     /VELS/
  RECORD /VELOCITY_VECTOR/ WRIST
  RECORD /VELOCITY_VECTOR/ BOD
  RECORD /VELOCITY_VECTOR/ CRF
  RECORD /VELOCITY_VECTOR/ MRF
  RECORD /VELOCITY_VECTOR/ KBRATE_BOD
  RECORD /JOINT/           VEL
ENDSTRUCTURE


STRUCTURE     /EXTERNALS/
```

```
      RECORD /VELOCITY_VECTOR/ HND
      RECORD /VELOCITY_VECTOR/ BOD
    ENDSTRUCTURE


    STRUCTURE     /FINGER_SENSOR/
      REAL        STRAIN(8)
    ENDSTRUCTURE


    STRUCTURE     /FT_SENSOR/
      LOGICAL     ZERO_FORCE
      LOGICAL     OVERLOAD
      LOGICAL     HEALTH
      INTEGER*4   ERROR_STATUS
      LOGICAL     TONCE
      RECORD /DATA_VECTOR/          FORTORQ_IN
      RECORD /DATA_VECTOR/          FORTORQ_OUT
      RECORD /COMMAND_VECTOR/  FORTORQ
      RECORD /GAIN_VECTOR/          KP
    ENDSTRUCTURE


    STRUCTURE     /RESOLVED_RATE_INPUTS/
      RECORD /VELOCITY_VECTOR/ EXTERNAL
      RECORD /EULER_VECTOR/         VISION
      RECORD /EULER_VECTOR/         FINGER
      RECORD /EULER_VECTOR/         WRIST
    ENDSTRUCTURE

    STRUCTURE     /OBJECT_DATA/
      RECORD /EULER_VECTOR/    mg_in_world
      RECORD /EULER_VECTOR/    cg_in_wrist
    ENDSTRUCTURE

C  Data Structures for a parallel-jaw end-effector
C
    STRUCTURE     /sensor_data/
      LOGICAL*1   desired        !  .TRUE. --> user wants sensor info
      INTEGER*4   xfire_ovl_lim  !  crossfire, overload, limit sensors
      INTEGER*4   proximity     ! proximity sensors
    END STRUCTURE     ! sensor_data
C
C   bit encoding for sensor information:  (bit 0 <--> 2**0)
C z-axis is parallel to axis of joint 6 of the manipulator
C y-axis is perpendicular to the jaw faces, directed toward right jaw
C x-axis is such that xyz form a right-handed coordinate reference
frame
C     bit #          proximity word       xfire_ovl_lim word
C     -----             --------------       ------------------
C    0                  Left jaw, +Ya          Crossfire
C    1                  Left jaw, +Z           Overload
C    2                  Left jaw, +X           Limit
```

```
C    3                  Left jaw, -X          Unused
C    4                  Right jaw, -Ya        Unused
C    5                  Right jaw, +Z         Unused
C    6                  Right jaw, +X         Unused
C    7                  Right jaw, -X         Unused
C    8-31               unused                unused
C
C
     STRUCTURE     /ratchet_data/
       LOGICAL*1   desired       ! .TRUE. --> user wants ratcheting
       REAL*4 center_gap    ! ratcheting centered at this gap (mm)
       REAL*4 sweep_degrees      ! total # degrees of ratchet sweep
       REAL*4 rate           ! speed of ratchet movement
     END STRUCTURE      ! ratchet_data
C
C
C
     STRUCTURE /pjg_end_effector/
       LOGICAL*1            initialize
       LOGICAL*1            null_motor
       REAL*4           jaw_gap
       REAL*4           error_gap
       REAL*4           target_gap
C A change in the "jaw_gap" value implies a commanded move
C to the given value.
       INTEGER*2            speed
c Speed contains the speed that the jaw is currently set to (via  'X')
       RECORD      /sensor_data/     sensors
       RECORD /ratchet_data/    ratchet
     END STRUCTURE

c
c Create all variables
c

     STRUCTURE     /MANIPULATOR/

       INTEGER*4 MODE_OUT          !CYBER control.
1=reset,2=hold,3=operate

       INTEGER*4 CTRL__MODE
       LOGICAL*1 DC_POWERON

       RECORD /ARM_DATA/         ARM
       RECORD /OBJECT_DATA/          OBJECT
       RECORD /CONTROL_STATION/ MANUAL
       RECORD /VISION_DATA/          VISION
       RECORD /FT_SENSOR/        WRIST
       RECORD /FINGER_SENSOR/        FINGER
       RECORD /PJG_END_EFFECTOR/     GRIPPER
       RECORD /VELS/             RESOLVED
       RECORD /EXTERNALS/        EXTERNAL
       RECORD /RESOLVED_RATE_INPUTS/  FLY
```

Appendix C
RTCOMMON Data Structure

```
      RECORD /HOMO_TRANS/       TRANSFORM
     ENDSTRUCTURE

     RECORD        /MANIPULATOR/          PUMA(2)

     RECORD        /P_BUFFER/         PUMA_BUFFER

     RECORD        /EULER_VECTOR/         ZERO

     RECORD        /HOMO_TRAN_MATRIX/     IDENTITY

C    Future cleanup project

     RECORD        /DATA_VECTOR/          GRAVITY_COMPED
     integer       DISPLAY_FORMAT


c Put all required variable in common
c
c    BYTE          FILLER(3958)
     INTEGER       ARM_SEL       !Selected arm

     COMMON        /RTCOMMON/
     1                  ZERO
     1                  ,IDENTITY
     3                  ,PUMA
     4                  ,DBUFFER
     4                  ,PUMA_BUFFER
     5                  ,arm_sel
     6                  ,arm_camera
     7                  ,camera_sel
     8                  ,frame_out
     9                  ,t
     1                  ,delt
     1                  ,csystem
     1                  ,rt$_status
     1                  ,gravity_comped
     1                  ,DISPLAY_FORMAT
c    4                  ,FILLER
```

BIBLIOGRAPHY

1. Cooper, E. G. and Sliwa, N. E. "A Rule-Based Script Generator for Control of Telerobotic Systems", NASA TP ???? (to appear 8/88).

2 Orlando, Nancy E. "An Intelligent Robotics Control Scheme," presented at the American Control Conference, June 6-8, 1984, San Diego, CA.

3. Paul, Robert. *Robot Manipulation: Mathematics, Programming and Control*, MIT Press, Cambridge, Mass., 1981.

4. Poe, Elmer C. and Goodwin, James C. *The S-100 and other Micro Buses*, Second Edition (1981), Howard W. Sams and Co., Inc., Indianapolis, Indiana, pp. 144-148.

5. Tella, Richard, Birk, John R., and Kelley, Robert B. "General Purpose Hands for Bin-picking Robots," *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-12, n. 6 , pp. 828-837.

6. Wise, M.J. "Description of the End-effector/Sensor System," Automation and Technology Branch Internal Report, NASA/Langley Research Center, Hampton, VA.

7. *MCS-51 Family of Single Chip Microcomputers – User's Manual*, July 1982, Intel Corporation, Santa Clara, California.

8. *Microcontroller Handbook*, 1986, pp. 7–1 through 8–85, Intel Corporation, Santa Clara, California.